JDBC en bref

Préparé par Saliha Yacoub

Table des matières

Introduction à JDBC

Introduction

JDBC, Java Data Base Connectivity est un ensemble de classes (API – Application Programming Interface --JAVA) permettant de se connecter à une base de données relationnelle en utilisant des requêtes SQL ou des procédures stockées.

L'API JDBC a été développée de manière à pouvoir se connecter à n'importe quelle base de données avec la même syntaxe; cette API est dite indépendante du SGBD utilisé.

Les classes JDBC font partie du package **java.sql** et **javax.sql**

JDBC permet entre autre :

1. L'établissement d'une connexion avec le SGBD.
2. L'envoi de requêtes SQL au SGBD, à partir du programme java.
3. Le traitement, au niveau du programme, des données retournées par le SGBD.
4. Le traitement des erreurs retournées par le SGBD lors de l'exécution d'une instruction.

Pilote de bases de données ou driver JDBC

- Un pilote ou driver JDBC est un "logiciel" qui permet de convertir les requêtes JDBC en requêtes spécifiques auprès de la base de données.
- Ce "logiciel" est en fait une implémentation de l'interface Driver, du package java.sql.

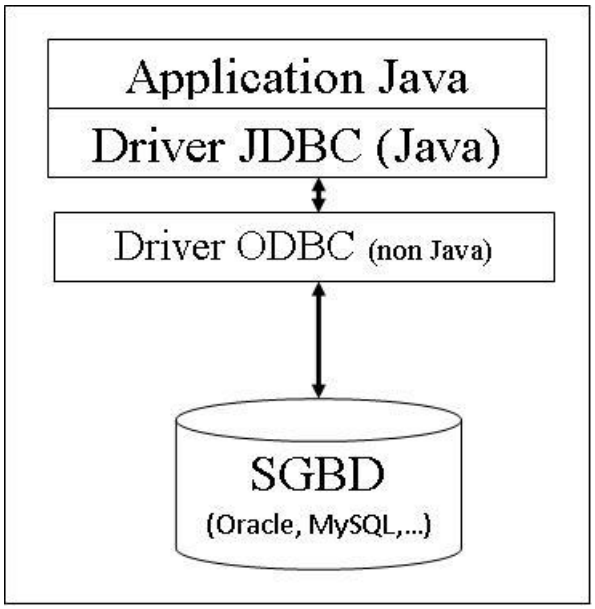
Dans le cas d'oracle, les drivers JDBC sont fournis par Oracle (en principe installés avec la base de données) téléchargeables à l'adresse.

http://www.oracle.com/technetwork/database/enterprise-edition/jdbc-112010-090769.html

Types de drivers JDBC

Il existe plusieurs types de pilotes JDBC

Les drivers de Type 1 : ODBC-JDBC bridges, ODBC (Open Data Base Connectivity) est une interface propre à Microsoft et qui permet l'accès à n'importe quelle base de données (Panneau de configuration /Outils d'administration/ Sources de données ODBC



Chaque requête JDBC est convertie par ce pilote en requête ODBC qui est par la suite convertie une seconde fois dans le langage spécifique de la base de donnée.

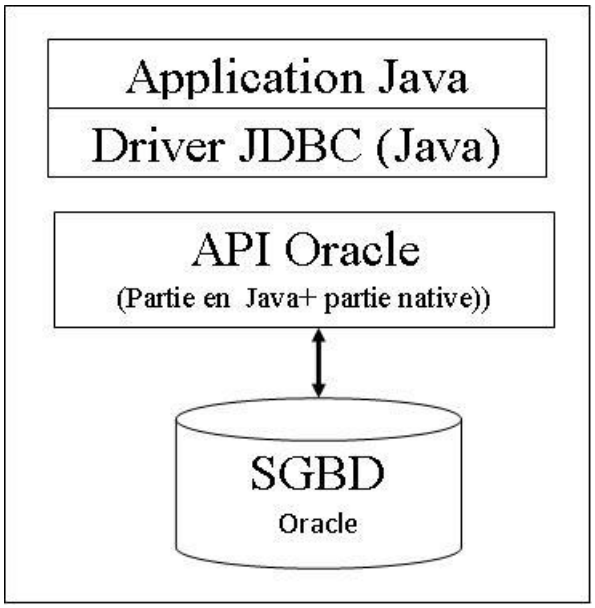
Cette technique est la moins optimale puisque les bases de données sont disponibles uniquement que par technologie ODBC.

Le SDK de Java fournit un pilote JDBC-ODBC :« sun.jdbc.odbc.JdbcOdbcDriver ».

Les drivers de Type 2

Ce type de driver traduit les appels de JDBC à un SGBD particulier, grâce à un mélange d'API java et d'API natives. (Propre au SGBD).

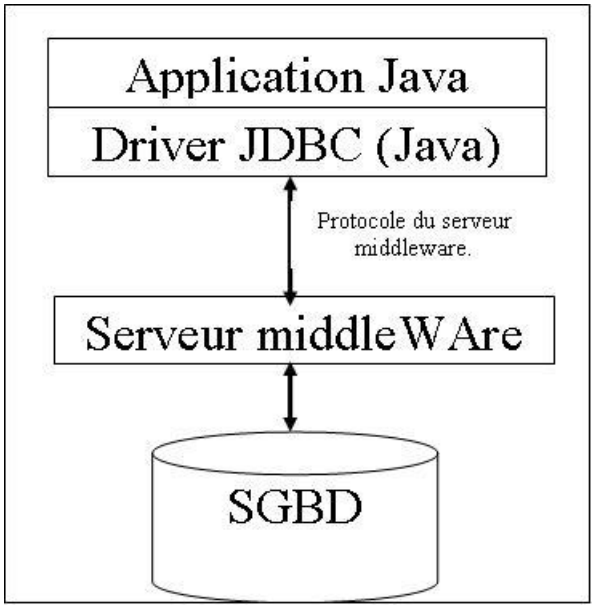
Ce Driver est fourni par l'éditeur de SGBD



Il est de ce fait nécessaire de fournir au client l'API native de la base de données.

Si on change le type de la base de données, on doit changer le pilote.

Drivers de type 3 (complètement écrit en JAVA)



Permet la connexion à une base de données via un serveur intermédiaire régissant l'accès aux multiples bases de données

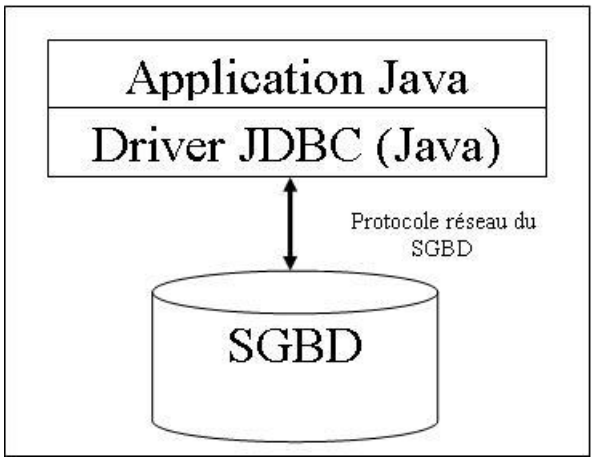
Ce type de driver est portable car écrit entièrement en java. Il est adapté pour le Web.

Cela exige une autre application serveur à installer et à entretenir.

Ce type de driver peut être facilement utilisé par une applet, mais dans ce cas le serveur intermédiaire doit obligatoirement être installé sur la machine contenant le serveur Web.

Drivers de type 4 (complètement écrit en JAVA)

Ce type de driver est connu sous le nom Direct Database Pure Java Driver), permet d'accéder directement à la base de données (sans ODBC ni Middleware). C'est le type le plus optimal.



C'est ce type de driver qui sera utilisé pour accéder aux bases de données oracle

Dans ce type de driver on retrouve le driver pour oracle (thin driver ou oracle.jdbc.driver.OracleDriver) dont le format de la chaine de connexion à une base de donnes est sous formes :jdbc:oracle:thin:@chainedeconnexion

Les drivers pour Oracle

Oracle supporte les drivers suivants :

Le JDBC Thin driver : c'est le driver par excellence. Il est de type 4, donc entièrement écrit en java. Il peut être utilisé dans les applications ou des applets. Pas besoin d'utres installation de logiciel oracle pour son fonctionnement. Il utilise le Oracle Net Services pour communiquer avec la base de données.(Oracle Net Services = ensemble de programmes permettant aux applications clientes de communiquer avec la base de données.

Oracle recommande d'utiliser le thin driver sauf s'il n'est pas supporté par l'application

L'URL de connexion incluant le type de river est :

url="jdbc:oracle:thin:@IP:PORT:ServiceName";

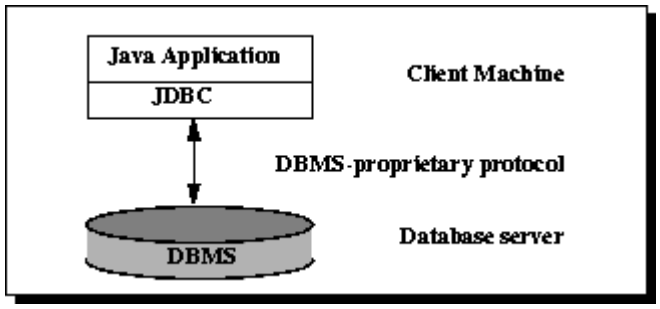
Le JDBC OCI (Oracle Call Inteface) , qui est un driver de type 2, écrit en Java et C et peut être utilisé uniquement avec des applications.

JDBC Server side thin Driver : identique au thin driver mais roule sur le serveur. Utilisé pour accéder à d'autres bases de données distantes.

Architecture

JDBC fonctionne selon les deux modèles suivants :

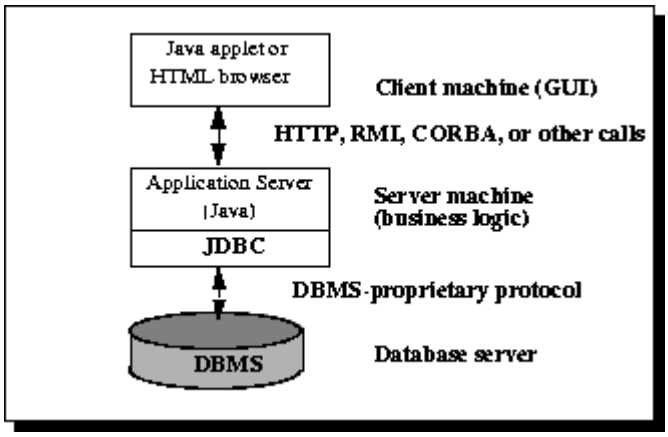
Modèle à deux couches (two-tier)



Dans le modèle two-tier, une application JAVA (ou une applet) dialogue avec le SGBD par l'intermédiaire du pilote JDBC. L'application JAVA et le pilote JDBC s'exécutent sur l'ordinateur client tandis que le SGBD est placé sur un serveur.

C'est ce type d'architecture qui nous concerne actuellement dans notre cours.

Modèles 3 couches (three-tier)



Dans le modèle three-tier, l'applet (ou l'application JAVA) ne dialogue plus directement avec un SGBD : un **middle-tier** fait le lien entre ces deux composants

Le SGBD exécute les requêtes SQL et envoie les résultats au middle tier. Ces résultats sont ensuite communiqués à l'applet sous forme d'appels http.

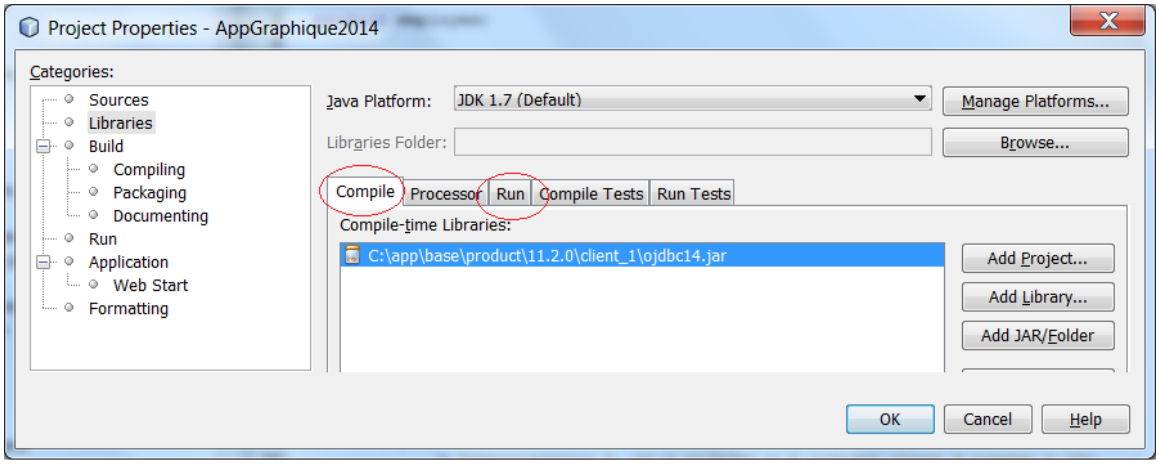
Fonctionnement

Tout programme JDBC fonctionne selon les étapes suivantes :

1. **Importer les packages nécessaires**
2. **Connexion à la base de données**
    i. Chargement du pilote de la BDD
    ii. Demande de connexion: s'identifiant auprès du SGBD et en précisant la base utilisée

3. **Traitement des commandes SQL**
4. **Traitement des résultats (Objet ResultSet)**
5. **Fermeture du RsultSet et du Statement**
6. **Mettre à jour la base de de données**
7. **Valider les mises à jour (COMMIT)**
8. **Fermeture de la connexion**.

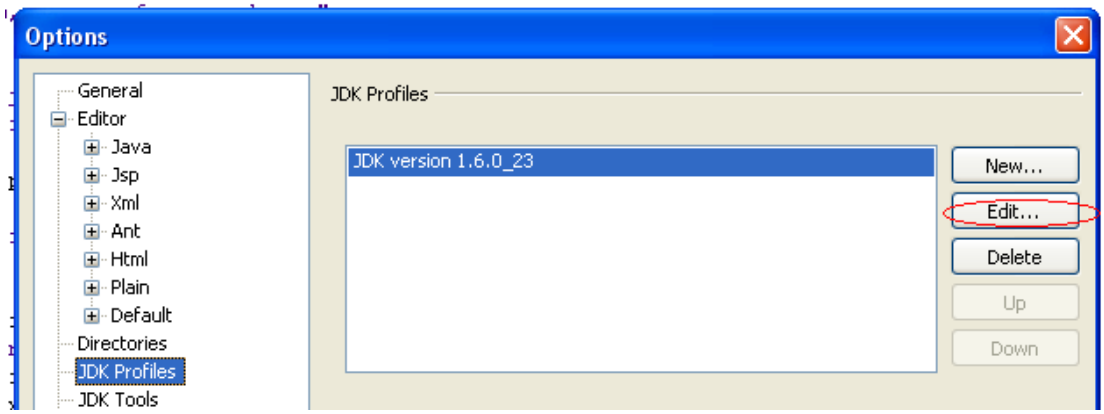
Pour utiliser JDBC avec NetBeans et oracle, vous devez inclure les librairies (.Jar) à votre projet : ces librairies sont, selon la version de votre JDK, ojdbc14.jar (pour JDK 1.6) .Pour ajouter ces librairies au projet cliquez sur bouton droit, puis propriétés



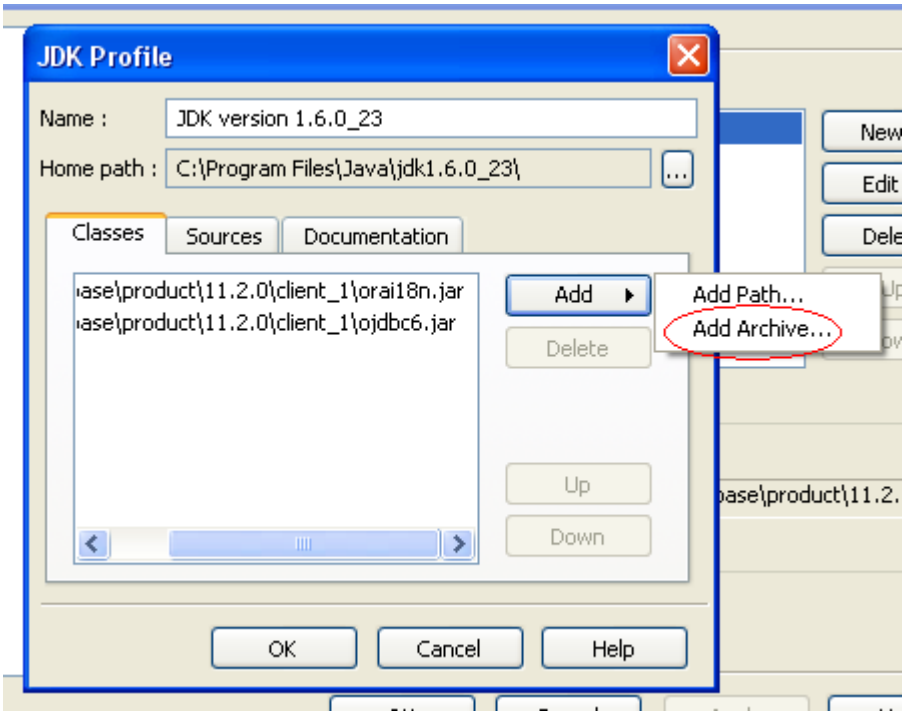
Ces librairies (pour oracle) sont dans le répertoire
C:\app\base\product\11.2.0\client_1

Pour utiliser JDBC avec Jcreator, il faut suivre les étapes :

Configure puis Options, choisir JDK Profiles, puis EDIT



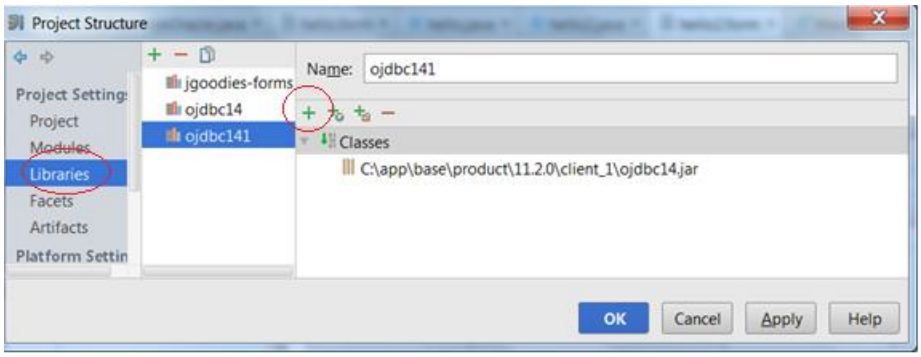
Puis Add Archive. La version actuelle est **ojdbc14.jar**



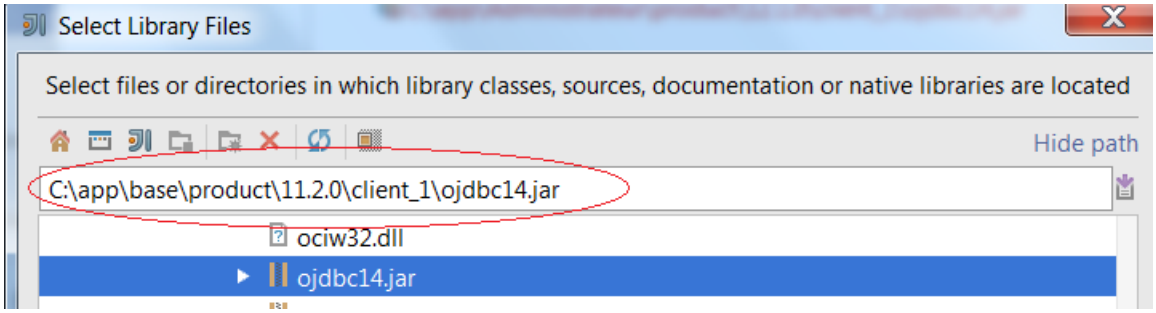
Les librairies pour Oracle sont dans **C:\app\base\product\11.2.0\client_1**

Pour utiliser JDBC avec Intellij Idea

Par le menu Fichier, Project Structure, à l'onglet Libraire, Ajouter



Puis, chercher ojdbc14.jar.



Établissement d'une connexion : versions du JDBC antérieurs à 4

1. **Importer le packages**
   import java.sql.*;
   import oracle.jdbc.driver.*;
   (d'autres packages seront nécessaires plus tard)
2. **Chargement du pilote (driver)**

Pour établir une connexion, il faut d'abord charger le driver en utilisant la méthode forName de la classe Class comme suit : **Class.forName(string driver)**. Pour oracle, l'instruction est la suivante :

```
Class.forName ("oracle.jdbc.driver.OracleDriver");
```

Quand une classe **Driver** est chargée, elle doit créer une instance d'elle même et s'enregistrer auprès du **DriverManager.**

La méthode **Class.forName(string driver)** fait partie du package **java.lang et** peut lancer une exception de type «ClassNotFoundException ».

Méthode 1 : Utilisation de la classe Class

Exemple1.java

```
package Exemple1;
import java.sql.*;
import oracle.jdbc.OracleDriver;

public class Exemple1 {

    public static void main(String[] args) {
        Connection conn = null;
        String user="user1";
        String mapasse ="user1";
        String url = "jdbc:oracle:thin:@205.237.244.251:1521:orcl";
        String sql1 ="select nomemp, prenomemp from employesclg where numemp =4";
        ResultSet rst;

        try    {
         Class.forName("oracle.jdbc.driver.OracleDriver");
          System.out.println("pilote chargé");
               }
        catch(ClassNotFoundException cnfe)
               {
        System.out.println("driver manquant");
               }
```

Méthode 2 : Utilisation du DriverManager.

On peut également demander à charger le driver auprès du DriverManager avec la méthode registerDriver comme suit

```
try  {
        DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
        System.out.println("pilote chargé");
      }
catch(SQLException sqldriver)
    {  System.out.println("driver manquant" );
    }
```

Remarquez que dans le premier cas, où on utilise la classe Class, le type d'exception renvoyée n'est pas SQL. C'est ClassNotFoundException. Pour la méthode forName, on passe le nom du Driver sous forme de chaîne caractère.

Dans le deuxième cas, le nom du driver est fourni directement à la méthode **registerDriver** est l'exception est de type SQL.

3. **Demander une connexion**
   Que l'on utilise la méthode 1 ou la méthode 2, une fois que le driver est chargé, il faudra demander une connexion.

   Cette connexion est obtenue grâce à la méthode **getConnection** de la classe DriverManager

   Cette méthode retourne la connexion qui est en fait, un **objet** implémentant l'interface «Connection».

   Connection connexion = DriverManager.getConnection(url); url désigne la chaine de connexion , dans le cas d'oracle la chaine de connexion est de forme :

   "jdbc:oracle:thin:@IP:port:orcl", "nomUsager","Motdepasse"

   Exemple :

```java
package Exemple1;
import java.sql.*;
import oracle.jdbc.OracleDriver;

public class Exemple1 {

    public static void main(String[] args) {
        Connection conn = null;
        String user="user1";
        String mapasse ="user1";
        String url = "jdbc:oracle:thin:@205.237.244.251:1521:orcl";

         try  {
                DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
                System.out.println("pilote chargé");
             }
         catch(SQLException sqldriver)
             {  System.out.println("driver manquant" );
             }
        // on ouvre une connexion avec les paramètres de connexion.
           try      {
            conn = DriverManager.getConnection(url,user,mapasse);
            System.out.println("vous êtes connectés");
                    }
          catch(SQLException sqlconnect){ System.out.println("connexion impossible");}
```

Toute connexion ouverte doit être fermée. La méthode Close de l'objet Connection permet de fermer une connexion.

```
import java.sql.*;
public class Exemple1 {

    public static void main(String[] args) {
        Connection conn = null;
        String user="user1";
        String mapasse ="user1";
        String url = "jdbc:oracle:thin:@205.237.244.251:1521:orcl";

         try  {
                DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
                System.out.println("pilote chargé");
             }
         catch(SQLException sqldriver)
             {  System.out.println("driver manquant" );
             }
        // on ouvre une connexion avec les paramètres de connexion.
          try      {
           conn = DriverManager.getConnection(url,user,mapasse);
           System.out.println("vous êtes connectés");
                  }
         catch(SQLException sqlconnect){ System.out.println("connexion impossible");}
          //on ferme la connexion
        finally
             {
              try
              {
              if (conn!=null)
                  conn.close();
              System.out.println("connexion fermée");
               }
              catch(SQLException se){}
             }
    }
}
```

Obtenir une connexion à une base de données Mysql

Mysql.java

```
package Mysql;
import java.sql.*;
public class Mysql {

    public static void main( String args[] ) {
        Connection conn = null;
        Statement stm = null;
        ResultSet rst=null;
        String ussr = "aaaaaa";
        String Bd = "jdbc:mysql://localhost:3306/depinfo2015";
        String mp = "aaaaa";
        String sql2 = "select nomgroupe from groupes where codegroupe ='inf1'";
        try {
            Class.forName("com.mysql.jdbc.Driver");

            System.out.println("Pilote Chargé");
             }
        catch (ClassNotFoundException e) {
            System.err.println(e.getMessage());
            System.exit(0);
             }
        try {
            conn = DriverManager.getConnection(Bd, ussr, mp);
            System.out.println("Pilote Chargé, Connexion ouverte");

            try{
                stm = conn.createStatement();
                rst = stm.executeQuery(sql2);
                if (rst.next())
                {
                    System.out.println("le nom du groupe est" + "   "+ rst.getString(1) );
                }
               }

            catch (SQLException sql)
            {
                System.out.println(sql.getMessage());
                System.exit(0);
            }
            finally {
                stm.close();
                rst.close();
                    }
             }
```

Suite du code : On complète avec le catch du try de connexion et le finally pour fermer la connexion

```
        catch (SQLException seq) {
            System.out.println(seq.getMessage());
            System.exit(0);
                }

        finally
        {
            try {
                if (conn != null)
                    conn.close();
                System.out.println("connexion fermée");

                }
            catch (SQLException se) {
            }
        }

    }
    }
```

Obtenir une connexion à une base de données Sqlite :

On ouvre une connexion, puis on exécute une requête de type SELECT.

SqLite.java ×

```
package Sqlite;
import java.sql.*;
public class SqLite {

    public static void main( String args[] )
    {
        Connection conn = null;
        Statement stm = null;
        ResultSet rst = null;
        String sql2 = "Select race from chats where nom ='Ruby'";

        try {
            Class.forName("org.sqlite.JDBC");

            System.out.println("Pilote Chargé");
             }
        catch ( ClassNotFoundException e )
            {
            System.out.println(e.getMessage());
            System.exit(0);
            }
        try {
            conn = DriverManager.getConnection("jdbc:sqlite:C:/Sqlite/BdLite.db;");
            System.out.println("Pilote Chargé, Connexion ouverte");

                    try{
                    stm = conn.createStatement();
                    rst = stm.executeQuery(sql2);
                    if (rst.next())
                    {
                   System.out.println("la race est" + "   "+ rst.getString(1) );
                    }

                    }

                    catch (SQLException sql)
                    {
                   System.out.println(sql.getMessage());
                   System.exit(0);
                       }
                   finally {
                    stm.close();
                   rst.close();
                    }
            }
```

Suite du code : On complète avec le catch du try de connexion et le finally pour fermer la connexion

```
        catch (SQLException sed )
        {
            System.out.println(sed.getMessage());
            System.exit(0);
        }

        finally
        {
            try {
                if (conn != null)
                    conn.close();
                System.out.println("connexion fermée");

            }
            catch (SQLException se) {
            }
        }


    }
}
```

Établissement d'une connexion : version 3 et plus du JDBC

Dans la version 4 du JDBC, il n'est pas nécessaire de charger explicitement le driver avec la méthode forName de la classe Class ou d'enregistrer explicitement le driver auprès du DriverManager.

Le DataSource permet d'utiliser un pool de connexion, qui est un mécanisme de réutilisation des connexions créées. Un pool de connexions ne ferme pas les connexions lors de l'appel à la méthode *close()*. Au lieu de fermer directement la connexion, celle-ci est "retournée" au pool et peut être utilisée ultérieurement. La gestion du pool est transparente pour l'utilisateur.

Un objet OracleDataSource définit un ensemble de méthodes permettant l'accès à la base de données Oracle.

Constructeur : OracleDataSource()

OracleDataSource ods = new OracleDataSource();

Méthode importantes :

getConnection()	Obtient une connexion à la base de données
setServerName	Définit le nom du serveur de la base de données utilisée. (ou son adresse IP)
setPortNumber	Définit le port du serveur de base s de données
setPassword	Définit le mot de passe avec lequel la connexion est obtenue.
setUser	Définit le username de l'usager
setServiceName	Définit le nom du service de la base de donnée : orcl
setDatabaseName	Définit le nom de la base de donnée : orcl
setURL	Définit la chaîne de connexion qui sera utilisée pour se connecter à la base de données. Elle est de la forme String url="jdbc:oracle:thin:@111.111.111.111:1521:orcl". Au lieu de définir l'IP , le protocole, le service name séparément, il est préférable d'utiliser l'URL.
getServerName	obtient le nom du serveur de la base de données utilisée. (ou son adresse IP)

getPortNumber	obtient le port du serveur de base s de données
getPassword	obtientt le mot de passe avec lequel la connexion est obtenue.
getUser	Obtient le username de l'usager
getServiceName	Obtient le nom du service de la base de donnée : orcl
getDatabaseName	Obtient le nom de la base de donnée : orcl

Pour les autres méthodes, veuillez consulter le site :

http://docs.oracle.com/cd/E11882_01/appdev.112/e13995/oracle/jdbc/pool/OracleDataSource.html

1. Importer les packages :

```
import java.sql.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.*;
```

2. La connexion elle-même

```
String user1 ="user1";
 String mdep ="oracle1";
 String url="jdbc:oracle:thin:@205.237.244.251:1521:orcl";

//déclarer un objet OracledataSource
OracleDataSource ods = new OracleDataSource();

// définir les paramètres de connexion pour l'objet OracleDataSourse ods
    ods.setURL(url);
    ods.setUser(user1);
    ods.setPassword(mdep);

// Appel de la méthode getConnection pour obtenir une connexion
Connection conn = ods.getConnection();
```

3. Exemple complet

```
package Exemple1;
import java.sql.*;
import oracle.jdbc.pool.*;
public class Exemple1 {

    public static void main(String[] args) {
        Connection conn = null;
        String user="user1";
        String mapasse ="user1";
        String url = "jdbc:oracle:thin:@205.237.244.251:1521:orcl";

    try {
        OracleDataSource ods = new OracleDataSource();
        ods.setURL(url);
        ods.setUser(user);
        ods.setPassword(mapasse);
        conn = ods.getConnection();
        System.out.println("vous etes connectés ");
        }

        catch(SQLException sqlods)
         {
         System.out.println("connexion impossible");
         }

        finally
             {
                try
                {
                if (conn!=null)
                    conn.close();
                System.out.println("connexion fermée");
                 }
                catch(SQLException se){}
             }

 }

}
```

Fermeture d'une connexion :

La connexion est fermée avec la méthode close de l'objet **connexion.close();**

Exécution de requêtas SQL : créer un « statement » d'une requête particulière.

Cette étape consiste à obtenir une **déclaration (zone de description de requête ou «*statement »)*** au travers de laquelle les requêtes SQL seront exécutées.

Il existe 3 types de déclarations:

1. Statement, instruction simple : permet d'exécuter directement et une fois l'action sur la base de données :

   Statement declaration1= connexion.createStatement();

2. PreparedStatement: instruction paramétrée. (cas des requêtes avec paramètres)

   - L'instruction est générique, des champs sont non remplis
   - Permet une précompilation de l'instruction optimisant les performances
   - Pour chaque exécution, on précise les champs manquants

   PreparedStatement declaration2= connexion.PreparedStatement (requetesql);

3. CallableStatement:

Une déclaration de type « CallableStatement» permet l'accès complet aux fonctions contenues dans la base de données.(cas des procédures stockées)

Exécution de requêtes SQL : executeUpdate; executeQuery, execute

A. **La méthode ExecuteUpdate** est utilisée pour les requêtes DML (INSERT, DELETE, UPDATE).

Cette méthode retourne le nombre de lignes affectées par la requête sql.

objetStatement.**executeUpdate**(String Requête_SQL]);

ou

objetPreparedStatement.**executeUpdate**(String Requête_SQL]);

Exemple:

```
package Exemple1;
import java.sql.*;
import oracle.jdbc.pool.*;
public class Exemple1 {

        public static void main(String[] args) {
        Connection conn = null;
        String user = "user1";
        String mapasse = "user1";
        String url = "jdbc:oracle:thin:@205.237.244.251:1521:orcl";
       String sqlIns = "insert into employesclg(numemp,
nomemp,prenomemp) values(25,'Alpha','Omega')";

        try {
            OracleDataSource ods = new OracleDataSource();
            ods.setURL(url);
            ods.setUser(user);
            ods.setPassword(mapasse);
            conn = ods.getConnection();
            System.out.println("vous êtes connectés ");

                    try
                       {
                       Statement stmins = conn.createStatement();
                       int n = stmins.executeUpdate(sqlIns);
                       System.out.println("nb de lignes ajoutée" + n);
                       }
                     catch (SQLException sqlinsertion)
                       {
                       System.out.println(sqlinsertion.getMessage());
                       }

                       finally
                       {
                       stmins.close();
                       }

        }
```

```
        catch (SQLException sqlods)
        {       System.out.println("connexion impossible");}



         finally {
            try {
                if (conn != null)
                    conn.close();
                System.out.println("connexion fermée");
            } catch (SQLException se) {
            }
        }
    }

}
```

B. **La méthode executeQuery,** permet d'exécuter une instruction SQL de type SELECT

Elle retourne un objet de type **ResultSet** contenant tous les résultats de la requête (les tuples sélectionnés).

Syntaxe

objetResultSet=objetStatement.**executeQuery(**String ordreSQL);

ou

objetResultSet=objetPreparedStatement.**executeQuery(**String ordreSQL);

Exploitation des résultats des requêtes SQL

L'interface ResultSet représente une table de lignes et de colonnes.

- Une ligne représente un enregistrement
- Une colonne représente un champ particulier de la table
- Un objet de type RsultSet possède un pointeur sur l'enregistrement courant. À la réception de cet objet, le pointeur se trouve devant le premier enregistrement.
- On y accède ligne par ligne, puis colonne par colonne dans la ligne.

Pour pouvoir récupérer les données contenues dans l'instance de **ResultSet**, celui-ci met à disposition des méthodes permettant de :

- Positionner le curseur sur l'enregistrement suivant avec la méthode next()

- public boolean **next();** Renvoi un booléen indiquant la présence d'un élément suivant.

- Accéder à la valeur d'un champ (par indice ou par nom) de l'enregistrement actuellement pointé par le curseur avec les méthodes getString(), getInt(), getDate() .....
  - public String getString(int indiceCol);
  - public String getString(String nomCol);
  - etc.

À la création du ResultSet, le curseur de parcours est positionné avant la première occurrence à traiter. **Le premier indice étant 1**

```
package Exemple1;
import java.sql.*;
import oracle.jdbc.pool.*;
public class Exemple1 {
         public static void main(String[] args) {
        Connection conn = null;
        String user = "user1";
        String mapasse = "user1";
        String url = "jdbc:oracle:thin:@205.237.244.251:1521:orcl";
         String sql1 = "select nomemp, prenomemp from employesclg ";
       Statement stm1 = null;
       ResultSet rst = null;
         try {
            OracleDataSource ods = new OracleDataSource();
            ods.setURL(url);
            ods.setUser(user);
            ods.setPassword(mapasse);
            conn = ods.getConnection();
            System.out.println("vous êtes connectés ");
               try {
                       stm1 = conn.createStatement();
                       rst = stm1.executeQuery(sql1);
                       while (rst.next())
                               {
                       //String nom1 = rst.getString(1);
                       //String prn1 = rst.getString(2);
                       String nom1 = rst.getString("nomemp");
                       String prn1 = rst.getString("prenomemp");
                       System.out.println( nom1  + "------" + prn1 );
                               }
                  }
               catch (SQLException sqlselect)
                       {
                        System.out.println(sqlselect.getMessage());
                       }
               finally {
               stm1.close();
```

```
                rst.close();
                        }


                } // premier try
        catch (SQLException sqlods)
        {       System.out.println("connexion impossible");}

         finally {
            try {
                if (conn != null)
                    conn.close();
                System.out.println("connexion fermée");
            } catch (SQLException se) {
            }
        }
    }

}
```

Remarque : Il est important de fermer le Statement et le ResultSet avec la méthode close() juste après leur utilisation pour libérer immédiatement les ressources qu'ils occupent. S'ils ne sont pas fermé par la méthode Close(),alors il le seront lorsque l'application sera fermées, mais ce n'est pas une façon recommandée.

C. **Méthode execute** (String ordre) : Retourne « **true** » si un résultat est disponible, « **false** » si non.

valeurbooléenne=objetStatement.**execute** (String ordre);

valeurbooléenne=objetPreparedStatement.**execute** (String ordre);

Exemple d'utilisation de la méthode Execute().

Dans le code suivant seul le contenu du try est indiqué.

Si la méthode Execute est utilisée alors, pour obtenir le ResultSet, il faut utiliser la méthode **getResultSet** du Statement (dans le cas qui nous concerne).

```
Statement stm = null;
ResultSet rst = null;
boolean nonvide;
String sql4 ="SELECT * FROM EMPLOYESBIDON";


stm =conn.createStatement();

nonvide = stm.execute(sql4);
 if(nonvide)
  {
      rst=stm.getResultSet();



      while (rst.next())
      {
          String nom1 = rst.getString(1);
          String prn1 = rst.getString(2);
          System.out.println( nom1  + "------" + prn1 );
      }


 }

// Suite du code ....
```

Utilisation du PreparedStatement

Ce type d'interface est utilisé pour des requêtes paramétrées. PreparedStatement est utilisé dans le cas où la requête va être exécutée plusieurs fois. De plus les requêtes sont précompilées.

Remarquez

1. Dans le requête le paramètre est représenté par **?**
2. Les paramètres sont passés dans l'ordre de leur présentation de la requête
3. Les paramètres et les valeurs sont passés comme suit :

- [Objet PreparedStatement].setString([index],[objet String]);
- [Objet PreparedStatement].setBoolean([index],[valeur]);
- [Objet PreparedStatement].setInt([index],[valeur]);
- [Objet PreparedStatement].setFloat([index],[valeur]);
- Etc…

4. La requête est exécuté par executeUpdate() ou executeQuery
5. On peut effacer le contenu des paramètres par la méthode ClearParameters()

Exemple : : Nous sommes déjà connétés.

```
conn = ods.getConnection();
String sql3 = "select nomemp, prenomemp from employesbidon where
emploi =?";

PreparedStatement stm = null;
ResultSet rst = null;

        try {
           stm = conn.prepareStatement(sql3);
            stm.setString(1,"ANALYSTE");
            rst = stm.executeQuery();
            while (rst.next())
            {
                String nom1 = rst.getString("nomemp");
                String prn1 = rst.getString("prenomemp");
                System.out.println( nom1  + "------" + prn1 );
            }
            stm.clearParameters();
            }
```

```
    catch (SQLException sqlselect)
     {
    System.out.println(sqlselect.getMessage());
     }
   finally {
         stm.close();
         rst.close();
           }
```

Type de parcours du ResultSet

Il est possible de parcourir le ResultSet de trois façons différentes selon le type de ce dernier. Par défaut, le parcourt est farward only.

ResultSet.TYPE_FORWARD _ONLY : accès séquentiel

ResultSet.TYPE_SCROLL_INSENSITIVE, accès direct sans modification (les occurrences ne reflètent pas les mises à jour qui peuvent intervenir durant le parcours) Permet de parcourir les résultats dans les deux sens.

1. Permet de parcourir les résultats dans les deux sens grâce aux méthodes:
    - Public boolean next();
    - public boolean previous();
    - Public boolean first();
    - Public boolean last();
2. Permet de connaitre la position courante du curseur à l'intérieur du ResultSet
    - Public boolean isBeforFirst();
    - public boolean isAfterLast();
    - Public boolean isFirst();
    - Public boolean isLast();

ResultSet.TYPE_SCROLL_SENSITIVE accès direct avec modification

Même principe que le type précédent sauf que, les occurrences reflètent les mises à jour qui peuvent intervenir durant le parcours

Modification des données du ResultSet:

Par défaut, un ResultSet contient des données en lecture seulement, mais il est possible d'obtenir un ResultSet modifiable.

ResultSet.CONCUR_READ_ONLY : lecture seule

ResultSet.CONCUR_UPDATABLE : mise à jour

Le type de ResultSet et le mode d'utilisation (read only ou updatable) doit se faire lors de la création du Statement ou du PreParedStatement

```
String sql3 = "select nomemp, prenomemp from employesbidon where
emploi =?";

PreparedStatement stm = null;
ResultSet rst = null;

        try {
           stm = conn.prepareStatement(sql3,
ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);

            stm.setString(1,"ANALYSTE");
            rst = stm.executeQuery();
            while (rst.next())
            {
                String nom1 = rst.getString("nomemp");
                String prn1 = rst.getString("prenomemp");
                System.out.println( nom1  + "------" + prn1 );
            }
            stm.clearParameters();
            }



    catch (SQLException sqlselect)
     {
    System.out.println(sqlselect.getMessage());
     }
   finally {
         stm.close();
         rst.close();
           }
```

Exemple

```
Statement stm2
=connexion.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
```

Dans le cas d'une modification (updatable), voici les opérations pour une mise à jour ou une insertion

Modifier la valeur du type et de la colonne donnée (par indice ou par nom) de l'enregistrement actuellement **pointé** :

- public void updateString(int indiceCol, String value);
- public void updateString(String nomCol, String value);
- public void updateInt(int indiceCol, Int value);
- public void updateInt(String nomCol, Int value);
- etc.

Appliquer dans la base de données les changements effectués sur l'enregistrement actuellement pointé : public void **updateRow**();

Exemple; (String sql2 = "select nom, prenom from employes";)

```
Statement stm4 =connexion.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
    ResultSet rst4=stm4.executeQuery(sql2);
    rst4.next();
    rst4.updateString("nom", "Coluche");
    rst4.updateString("prenom", "Moses");
    rst4.updateRow();
    connexion.commit();
```

Dans le cas d'une insertion :

Aller sur un emplacement vide permettant d'insérer un nouvel enregistrement : public void **moveToInsertRow**();

Insérer dans la base de données l'enregistrement actuellement pointé : public void **insertRow**();

Exemple; (String sql2 = "select nom, prenom from employes";)

```
Statement stm3 =connexion.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
      ResultSet rst3=stm3.executeQuery(sql2);
      rst3.moveToInsertRow();
      rst3.updateString("nom", "Samuel");
      rst3.updateString("prenom", "Yacoub");
      rst3.insertRow();
      connexion.commit();
```

Méthode de déplacement dans le ResultSet

Pour se déplacer à l'intérieur du ResultSet, on utilisera la méthode **next()** pour le parcours avant et la méthode **previous()** pour le parcours inverse. Les autres méthodes sont données dans le tableau suivant.

Méthode	**Rôle**
boolean isBeforeFirst()	booléen qui indique si la position courante du curseur se trouve avant la première ligne
boolean isAfterLast()	booléen qui indique si la position courante du curseur se trouve après la dernière ligne
boolean isFirst()	booléen qui indique si le curseur est positionné sur la première ligne
boolean isLast()	booléen qui indique si le curseur est positionné sur la dernière ligne
boolean first()	déplacer le curseur sur la première ligne
boolean last()	déplacer le curseur sur la dernière ligne
boolean absolute()	déplace le curseur sur la ligne dont le numéro est fourni en paramètre à partir du début si il est positif et à partir de la fin si il est négatif. 1 déplace sur la première ligne, -1 sur la dernière, -2 sur l'avant dernière ...
boolean relative(int)	déplacer le curseur du nombre de lignes fourni en paramètre par rapport à la position courante du curseur. Le paramètre doit être négatif pour se déplacer vers le début et positif pur se déplacer vers la fin. Avant l'appel de cette méthode, il faut obligatoirement que le curseur soit positionné sur une ligne.
boolean previous()	déplacer le curseur sur la ligne précédente. Le booléen indique si la première occurrence est dépassée.

Boolean next()	déplacer le curseur sur la ligne suivante. Le booléen indique si la dernière occurrence est dépassée.
void afterLast()	déplacer le curseur après la dernière ligne
void beforeFirst()	déplacer le curseur avant la première ligne
int getRow()	renvoie le numéro de la ligne courante

Source :http://docs.oracle.com/javase/7/docs/api/java/sql/ResultSet.html

Méthodes pour obtenir les données et la structure

Méthode	**Rôle**
getInt(int)	retourne le contenu de la colonne dont le numéro est passé en paramètre sous forme d'entier.
getInt(String)	retourne le contenu de la colonne dont le nom est passé en paramètre sous forme d'entier.
getFloat(int)	retourne le contenu de la colonne dont le numéro est passé en paramètre sous forme de nombre flottant.
getFloat(String)	
getDate(int)	retourne le contenu de la colonne dont le numéro est passé en paramètre sous forme de date.
getDate(String)	
next()	se déplace sur le prochain enregistrement : retourne false si la fin est atteinte
Close()	ferme le ResultSet
getMetaData()	retourne un objet ResultSetMetaData associé au ResultSet.

Type de données JDBC (correspondance SQL et JAVA)

Type SQL	Méthode ResultSet	Type Java
ARRAY	getArray	java.sql.Array
BIGINT	getLong	long
BINARY	getBytes	byte[]
BIT	getBoolean	boolean
BLOB	getBlob	java.sql.Blob
CHAR	getString	java.lang.String
CLOB	getClob	java.sql.Clob
DATE	getDate	java.sql.Date
DECIMAL	getBigDecimal	java.math.BigDecimal
DINSTINCT	getTypeDeBase	typeDeBase
DOUBLE	getDouble	double
FLOAT	getDouble	double
INTEGER	getInt	int
JAVA_OBJECT	(type)getObject	type
LONGVARBINARY	getBytes	byte[]
LONGVARCHAR	getString	java.lang.String
NUMERIC	getBigDecimal	java.math.BigDecimal
OTHER	getObject	java.lang.Object
REAL	getFloat	float
REF	getRef	java.sql.Ref
SMALLINT	getShort	short
STRUCT	(type)getObject	type

TIME	getTime	java.sql.Time
TIMESTAMP	getTimestamp	java.sql.Timestamp
TINYINT	getByte	byte
VARBINARY	getBytes	byte[]
VARCHAR	getString	java.lang.String

Autres informations du Resultset :

Le ResultSet présente une interface qui permet d'avoir de l'information sur la structure des données qu'il contient. Cette interface est : **ResultSetMetaData**

- Obtenir le "metadata" avec la méthode **getMetaData() du resultSet**

Exemple:

```
ResultSetMetaData colonnes = rst.getMetaData();
```

- Obtenir le nombre de colonnes d'un objet de type ResultSet: avec la méthode **getColumnCount();**

  Exemple :

```
int nbcolonne = colonnes.getColumnCount();
```

- **obtenir le nom des colonnes d'indice connu.**

getColumnName(int [indice colonne]);

Exemple

```
colonnes.getColumnName(2)
```

- **obtenir le type d'une colonne avec la méthode getColumnTypeName(int [indice colonne]);**

```
colonnes.getColumnTypeName(3)
```

Exemple :

```
String sql4 ="SELECT * FROM EMPLOYESBIDON";
Statement stm = null;
ResultSet rst = null;
boolean nonvide;
Connection conn = null;

// après la connexion
try {

Statement stmx = conn.prepareStatement(sql3,
ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);
       stm =conn.createStatement();
       nonvide = stm.execute(sql4);
             if(nonvide)
               {
                rst=stm.getResultSet();
                //Afficher les metadonnées ici
                ResultSetMetaData colonnes = rst.getMetaData();
                int nbcolonne = colonnes.getColumnCount();

              for(int i=1; i<=nbcolonne;i++)
                   {
System.out.print(colonnes.getColumnName(i)+ "\t\t");
       System.out.print(colonnes.getColumnTypeName(i)+ "\t\t");
                    }
                            System.out.println();
                        //Afficher le resultSet
                            while (rst.next())
                            {
                                String nom1 = rst.getString(1);
                                String prn1 = rst.getString(2);
                                System.out.println( nom1+ prn1 );
                            }

                }
             }

             catch (SQLException sqlselect)  {
                 System.out.println(sqlselect.getMessage());
                                          }
             finally {
                      stm.close();
                      rst.close();
                     }
```

Sources :

http://en.wikipedia.org/wiki/JDBC_driver#Type_4_Driver_-_Native-Protocol_Driver

http://download.oracle.com/javase/tutorial/jdbc/overview/architecture.html

http://jguillard.developpez.com/JDBC/11.html

http://java.developpez.com/

http://docs.oracle.com/javase/7/docs/api/java/sql/ResultSetMetaData.html

http://download.oracle.com/javase/1.4.2/docs/api/java/sql/package-summary.html
pour le package Java.qsl

http://docs.oracle.com/cd/B14099_19/web.1012/b14017/jdbcejb.htm#i1001657

https://www.jetbrains.com/idea/help/gui-designer-basics.html

Introduction à JDBCde Denis Brunet